Modern Yazılım Mühendisliğinde Scrum Paradigmasının Eleştirel Analizi: Sprint Döngüleri, Akış Kesintileri ve DevOps Hızı Üzerine Bir İnceleme

Mustafa Güven

Yazılım yönetimi, insanları yönetmeye odaklanmak yerine sistemi ve akışı tasarlamaya yönelmedikçe, kutsallaştırılmış ezberlerin ve amaçların önüne geçen yöntemlerin içinde savrulmaya mahkumdur.

Gerçek Scrum Bu Değil[1]

Herhangi bir şeyin yapısal sorunları tartışmaya açıldığında, sıklıkla başvurulan savunma stratejilerinden biri Gerçek X bu değil önermesidir. Literatürde No True Scotsman safsatası olarak da bilinen bu yaklaşım, hedeflenen kavramı tüm somut eleştirilerden izole ederek onu soyut, kusursuz ve hatta münezzeh[2] bir ideal düzlemine taşır. Bu argüman sayesinde kavram, uygulama aşamasında ortaya çıkan hiçbir hata veya başarısızlıkla ilişkilendirilemez hale getirilir. Fakat gözden kaçırılan şey aslında bu durumun, ilgili konsepti eleştirilemez kıldığı ölçüde onu gerçekliğin de denetiminden kopardığı yani fikri savunuyormuş gibi görünse de aslında onu yaşayan dünyadan ayrıştırıyor olmasıdır.

Bu tür idealleştirmeler, kavramın pratik değerini ve geliştirilebilirliğini ortadan kaldıran entelektüel tıkanıklığa yol açar. Bir teori veya modeli, sahadaki başarısızlıklarından tamamen muaf tutmak, o modelin gerçek dünya ile kurduğu geri bildirim döngüsünü koparmak demektir. Eğer ideal olan hiçbir zaman hayata geçirilemiyorsa, o kavram bir çözüm önerisi olmaktan çıkıp statik bir dogmaya dönüşmüştür. Oysa ki bir sistemin veya düşüncenin rasyonalitesi kağıt üzerindeki mükemmelliği kadar karşılaştığı krizlere verdiği somut yanıtlar ve yanlışlanabilirliği ile de ölçülmelidir. Yani gerçeklik, soyut bir idealin yanında somut çıktılarla da tanımlanabildiğinde ölçülebilir değerlendirme yapabilmeyi mümkün kılar.

Bu çalışmada, scrum paradigmasının merkezinde yer alan sprint kavramının eleştirel bir analizini sunmaya çabalarken, Gerçek Scrum bu değil argümanı ile dışsallaştırılmaya çalışılmasını, teori ile pratik arasındaki tutarsızlığı tartması adına okuyucunun takdirine ve muhakemesine bırakıyorum. Zira bir metodolojinin başarısı, ancak başarısızlıklarının sorumluluğunu üstlenebildiği ve gerçek dünya kısıtlarıyla yüzleşebildiği ölçüde meşruiyet kazanır. Temel amacım, okuyucunun kendi deneyim süzgecinden geçirebileceği bu yüzleşme ortamını oluşturmaktır.

Tabulaşan Sprint ve Dogmatikleşen Çeviklik

Nedense yazılım geliştirme süreçlerindeki tıkanıklıklar, değer teslimatı (value delivery) sırasındaki aksaklıklar veya ekiplerin yaşadığı tükenmişlik sendromları (bu en bariz retrospektiflere iştirakteki isteksizlikte görülür) hiç scrum metodolojinin yapısal sınırlarına dayandırılmaz da bunun yerine sprint ritüellerinin kitaba uygun yapılmamasına bağlanır. Ve yine nedense karmaşık sistem mimarileri ve yazılım geliştirme yaşam döngüsünün gerçeklerinden bağımsız her organizasyonel oluşumda scrum'ın dayattığı ritüellerin muhakkak çalışacak evrensel doğrular olduğunu kabul etmemiz beklenir.

Üstelik yazılım geliştirme ekosistemi, özellikle Modern DevOps iş yapış şekilleri üzerinden son yıllarda çeviklik (agility) kavramı etrafında köklü bir dönüşüm geçirmiş ve bu dönüşüm ekiplere dayatılan scrum disiplininde hiç farklı başkalaşımlara uğratılmamışken. Her yerde aynı şeyleri görüyoruz, dinliyoruz.

Oysa modern yazılım mühendisliği pratikleri, özellikle trunk based development (TBD) ve continuous delivery (CI/CD) gibi disiplinler, kodun kesintisiz bir akış halinde olduğu, her an yayına hazır dinamik bir yapıyı zorunlu kılar. Buna karşın scrum'ın temel yapı taşı olan sprint mantığı, doğası gereği işi, yığınlar (batch) halinde işlemeyi öngören, zaman kesitlerine dayalı (time windowed) ve dur kalklı bir süreçtir. Bu çelişki, yazılım geliştirme döngüsünü bekleme sürelerine hapsederken aynı zamanda teslim süresi (lead time), döngü süresi (cycle time) ve pazara sunma süresi (time to market) metriklerini de kaçınılmaz olarak uzatır. Ve bu noktada siz, neden müşteriye sunulan değer üretiminde teorik kapasitenizin çok altında kaldığınızı, hız (velocity) grafiklerindeki artışa rağmen gerçek iş çıktısında (outcome) neden anlamlı bir ivme yakalayamadığınızı bir türlü rasyonel bir zemine oturtamazsınız. [3] [4] [5]

Ekipçe harcanan büyük bir efor vardır ama bu eforun ürüne gideceğine bizzat sürecin kendisine gittiğini fark ettiğinizde optimize ettiğiniz şeyin değer akışı (value stream) olmadığını bunun yerine süreç bürokrasisi olduğunu anlarsınız. Bu noktada görürsünüz ki akış verimliliği (flow efficiency) dramatik şekilde düşüyordur, çünkü mühendislik kapasitesi konsantre bir biçimde kod üretmek veya sorunları çözmek yerine, bitmeyen afaki tahminleme (estimation) pazarlıklarına, toplantılara, ritüellere, yani işi yönetme işine (work about work) harcanmaktadır. Sonuç, sürekli meşgul görünen ama dağıtım sıklığı (deployment frequency) düşük, hantal bir takımdır. Bunu ilerideki dora metrik ve elite takımlar bölümünde daha iyi gözlemleyeceğiz.

Modern Yazılım Mühendisliğinin Doğası ve Hız Paradigması

Günümüzde yazılım mühendisliği sadece kod yazmaktan ibaret değil. Karmaşık sistemlerin sürekli evrimi ve üretilen değerin anlık sunumu artık bu mühendislik dalının en belirgin olgunluk göstergesi. Hatta unutmamak gerekir bu evrimin başatlarından DevOps paradigması, yazılım geliştirme ve operasyon arasındaki sınırları kaldırarak, fikirden üretim ortamına (production) giden yolu en aza indirme düşüncesi üzerine inşa edildi. Hız eskiden bir tercihti artık yazılım mühendisliğinin zorunluluğu oldu. Peki durum bu iken sprint gerçekten doğru tercih mi?

Scrum'ın iteratif yapısı, teslimatı sprint sonlarına endeksleyerek yapay bir gecikme (latency) yaratır. Yazılım geliştirme ekipleri, sprint planlaması sırasında kapasitelerini bir zaman dilimi (timebox) içine sığacak görevlerle doldururlar. Bu, işin biriktirilmesi ve belirli bir periyotta toplu olarak bitmiş sayılması anlamına gelen bir yığın işleme

[1] https://en.wikipedia.org/wiki/No_true_Scotsman
[2] Eksiklik, noksanlık, kötülükten arındırılmış olan
[3] Why We've Ditched Scrum Sprints, Product Coalition, https://medium.productcoalition.com/why-weve-ditched-scrum-sprints-and-you-should-too-cdf5678e5199
[4] Time-boxed Sprints vs Process Flow: What's Right for Your Team?, Agile Sherpas, https://www.agilesherpas.com/blog/sprints-vs-process-flow
[5] How Sprinting Slows You Down: A Better Way to Build Software, https://www.everlaw.com/blog/ediscovery-software/how-sprinting-slows-you-down/

(batch processing) modelidir. Oysa modern veri işleme ve mühendislik paradigması, işin sisteme girdiği anda değer üretmeye başladığı akış (flow) modellerine doğru evrilmektedir.[6] Scrum'ın sprint mantığı, her iki haftada bir durup yeniden başlama ritmiyle, bu sürekli akışa zarar verir.

Metrik	Scrum	Continuous Flow
Unit of Work (Teslimat Birimi)	Sprint Hedefi	Tekil ve Atomik Görevler (Single Piece Flow)
Feedback Loop	İterasyon Sonu (Ort 4 Hafta)	Anlık ve Sürekli (Saatler)
Commit, Merge, Review Sıklığı	Başta neredeyse hiçbir şey yoktur, sprint sonuna doğru yoğunlaşır.	Günde en az 1 kere ve çoğunlukla birden fazla (TBD)
Risk Yönetimi	İterasyon sonunda toplu kontrol	Her commit sonrası automated test
Response to Change[7] (Çeviklik Metriği)	Bir sonraki sprint planlaması	Anlık (Just in Time) Önceliklendirme, Pull based takvimlendirme

Tabloda görüldüğü üzere, Scrum ve Continuous Flow arasındaki temel fark, zamanın ve işin nasıl yönetildiğinde yatmaktadır. Scrum'da zaman, işi kısıtlayan bir kesit iken, modern DevOps yaklaşımlarında zaman, değerin aktığı kesintisiz bir eksendir.

TBD ve Sprint Çatışması

TBD, yazılım mühendisliği ekiplerinin tüm değişiklikleri doğrudan main branch'e (trunk) gönderdiği ve uzun süreli feature branchlerinden kaçındığı bir modeldir. Bu çerçevede TBD, CI'ın kalbidir çünkü kod ne kadar hızlı birleşirse (merge), entegrasyon hataları o kadar erken tespit edilir. Ancak scrum'ın sprint yapısı, ekipleri genellikle sprint sonuna yetiştirme motivasyonuyla, kodları kendi branchlerinde saklamaya ve büyük bir merge operasyonunu sprint sonuna bırakmaya iter.

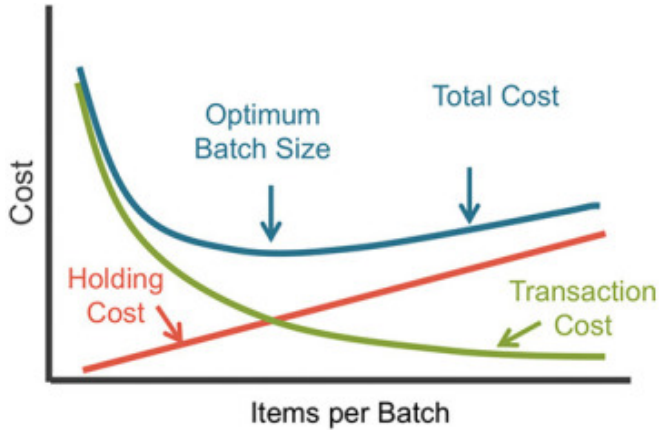
Sprint planlama sırasında seçilen görevler, genellikle iki haftalık bir süre zarfında tamamlanacak şekilde paketlenir (bu süre 3 hafta da olabilir 4 hafta da ama genelde 2 hafta büyük çoğunluğun kabulüdür). Bu durum, geliştiricilerin kodlarını main branch'e merge etmeleri yerine, task bitene kadar izole branchlerde çalışmalarına neden olur. Araştırmalar, branchlerin ömrü uzadıkça, birleştirme (merge) maliyetinin ve conflictlerin dramatik şekilde arttığını göstermektedir. [8]
Bununla birlikte TBD ise hızlı ve kısa süreli commitleri savunur. Sprint, bu doğal ritmi bozarak, geliştiricileri teslimat tarihlerine uymak için teknik borç biriktirmeye veya entegrasyonu geciktirmeye zorlar. TBD'nin başarısı için gerekli olan feature flag ve soyutlama ile dallanma (Branch by Abstraction) gibi teknikler, kodun üretime (production) gönderilmesi (deployment) ile son kullanıcıya açılması (release) arasındaki bağı koparır.[9] Bu ayrım, Scrum'ın her sprint sonunda sunulabilir ürün artışı (shippable increment) tanımını geçersiz kılmaktadır. Çünkü mühendislik ekibi her saat başı üretime kod gönderebiliyorsa, 2 haftalık artış beklemek sadece özünde saçma bürokratik engelden başka bir şey değildir. [10]

Kuyruk Teorisi ve İstatistiksel Verimlilik

Yazılım geliştirme süreçleri, istatistiksel olarak birer kuyruk sistemi (queuing system) olarak modellenebilir. Matematikçi John Little tarafından formüle edilen Little Yasası[11], bir sistemdeki ortalama iş miktarının (L), sisteme giriş hızı (λ) ve sistemde geçirilen süre (W) ile doğrudan ilişkili olduğunu belirtir.

$$L=\lambda W$$

Bu yasa, yazılım mühendisliğinde lead time (teslim süresi) ve throughput (çıktı hızı) arasındaki dengeyi anlamak için kritiktir. Scrum ekipleri, sprint planlamasında genellikle kapasitelerini tam doldurmayı (full utilization) hedeflerler. Ancak kuyruk teorisi, bir sistemde kullanım oranı %100'e yaklaştığında, bekleme sürelerinin (W) doğrusal değil, eksponansiyel olarak arttığını iddia eder. [12]



Bekletme Maliyeti (Holding Cost)
Yazılan kodun canlıya çıkmadan, kullanıcıya değer üretmeden kenarda beklemesinin maliyetidir. Sprint bu maliyeti artırır.

İşlem Maliyeti - Deployment (Transaction Cost)
O kodu (değeri) test etme, derleme ve canlıya alma (deployment) eforudur.

Yukarıda gördüğümüz Donald Reinertsen[13]'in meşhur U eğrisi bize şunu söyler: İşleri büyük paketler halinde biriktirmenin zararı (bekletme maliyeti) ile, her küçük parçayı tek tek işlemeye çalışmanın zorluğunu (işlem maliyeti) dengeleyip toplam maliyeti en aza indiren ideal iş büyüklüğünü bulmadan optimum akış verimliliği (flow efficiency) ve değer teslimat hızına (value delivery rate) ulaşılamaz. Bunu gelin bir analojiyle basitleştirelim, canınız her yumurta çektiğinde sadece 1 adet yumurta almak için arabaya binip markete giderseniz sürekli yolda zaman kaybeder, benzin harcarsınız. Yani paketleri çok küçük tutarsanız (her işi tek tek yaparsanız), işi yapma maliyetiniz tavan yapar (grafikteki

[6] Batch Processing vs Stream Processing: Key Differences for 2025, Atlan, https://atlan.com/batch-processing-vs-stream-processing/
Batch Vs Stream Processing | System Design | AlgoMaster.io, https://algomaster.io/learn/system-design/batch-vs-stream-processing.
Batch Processing vs. Stream Processing: A Comprehensive Guide, Rivery, https://rivery.io/blog/batch-vs-stream-processing-pros-and-cons-2/
[7] Responsiveness (Yanıt Verebilirlik), Adaptability (Uyarlanabilirlik), Reprioritization Latency (Yeniden Önceliklendirme Gecikmesi)
[8] Branching is Killing Your Delivery, b-agile, https://www.bagile.co.uk/branching-killing-delivery-trunk-based-development/
Trunk-Based Development: An Overview and Implementation Guide, Axify, https://axify.io/blog/trunk-based-development

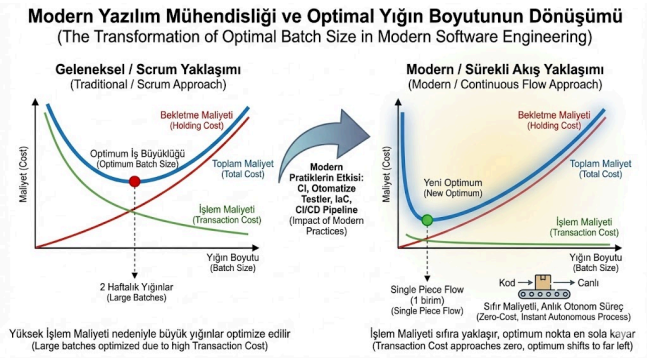
[9] Trunk-Based Development (TBD) vs Git Flow | by Julien Sanchez-Porro, Medium, https://medium.com/vield-studio/trunk-based-development-tbd-vs-git-flow-b73bb110452d
[10] How Big Tech Runs Tech Projects and the Curious Absence of Scrum, Reddit, https://www.reddit.com/r/programming/comments/pupuxl/how_big_tech_runs_tech_projects_and_the_curious/
[11] https://en.wikipedia.org/wiki/Little%27s_law
[12] Principle #6: Visualize and limit WIP, reduce batch sizes, and manage queue lengths, https://www.informit.com/articles/article.aspx?p=2928185&seqNum=6
[13] The Principles of Product Development Flow: Second Generation Lean Product Development, January 1 2009, Donald G. Reinertsen

yeşil çizgi). Markete gitmek çok yorucu, o yüzden 1 yıllık yiyeceği tek seferde alayım derseniz de bu sefer başka bir sorun başlar, yiyecekler bozulur, buzdolabında yer kalmaz, paranızı tek seferde o erzağa bağladığınız için nakit sıkıntısı çekersiniz, vb. Yani işleri devasa paketler halinde yaptıkça da elinizde tutmanın zararı tavan yapar (grafikteki kırmızı çizgi). Biliyoruz ki optimum olan, ne her yumurta için markete gitmek, ne de 1 yıllık erzağı tek seferde almak. U eğrisi bunu hesaplamak için iki maliyeti topla (Total Cost / Mavi Çizgi) ve o eğrinin en altındaki çukur noktayı bul der. O çukur nokta senin haftalık ideal alışveriş zaman aralığın olacaktır.

Arabayla Markete Gitmek vs Işınlanma Makinesi

Bu noktada haklı olarak şu itiraz yükselebilir, peki TBD veya CD pratiklerinde her commit'i canlıya almak, her yumurta için ayrı ayrı markete gitmek demek değil midir? Geleneksel proje yönetimi paradigmalarında evet, öyledir. Çünkü scrum veya waterfall gibi metodolojiler, test, entegrasyon ve deployment süreçlerinin manuel, risklerle dolu, tehdit içerici, acı verici ve yüksek maliyetli olduğunu varsayar. Ancak modern yazılım mühendisliğinin gözden kaçırılan sırrı, U eğrisindeki grafiğin sabit olmamasıdır.[14]

CI, otomatize edilmiş testler (automated tests) ve Infrastructure as Code (IaC) gibi pratikler, sistemdeki işlem maliyetini (grafikteki yeşil çizgi) dramatik bir biçimde aşağı çeker. Yazılım mimarisinde bir cicd pipeline inşa ettiğinizde, markete gitme eylemini sıfır maliyetli ve anlık bir otonom sürece dönüştürmüş olursunuz. İşlem maliyeti (transaction cost) sıfıra yaklaştığında, matematiksel olarak toplam maliyet eğrisinin (mavi çizgi) dip noktası grafiğin en soluna kayar. Artık optimum iş büyüklüğü (batch size) 2 haftalık yığınlar değildir sadece 1 birimdir (Single Piece Flow). İşlem maliyetinin olmadığı bir dünyada, kodu bekletmenin (holding cost) getirdiği riskler (merge conflictler), geri bildirim gecikmeleri katlanılamaz hale gelir. İşte bu yüzden her committe markete gitmek mantıksız değil, aksine modern platform mühendisliğinin ulaştığı en rafine verimlilik noktasıdır.



Aslında bunu da yine daha sade bir dille şöyle izah etmeye çalışalım. Evet, fiziksel dünyada her yumurta için markete gitmek mantıksızdır çünkü işlem maliyeti (transaction cost) sabittir ve yüksektir (yol, benzin, zaman, vb). Scrum da işte tam olarak bu fiziksel dünya kısıtlamalarına göre tasarlandığı, yani derleme, test etme ve canlıya alma süreçlerinin (markete gitmenin) maliyetli ve acı verici olduğunu varsaydığı için bunların hepsi zor iş, 2 haftalık yığınlar üzerinden ilerlemek en mantıklısı der. Peki ya bu analojiyi şöyle değiştirirsek? Ya mutfağınızda doğrudan markete açılan, bedavaya çalışan ve saniyeler içinde yumurtayı getiren bir ışınlanma makineniz veya otonom bir drone sisteminiz olsaydı? Eğer markete gitmenin maliyeti (zaman ve efor olarak) sıfıra düşseydi, o zaman evde 2 haftalık yumurta stoklamak (bekletme maliyeti) dünyanın en saçma işi haline gelirdi değil mi? Mesela taze taze, her ihtiyacınız olduğunda birer tane almak varken?

Görüldüğü gibi sprint taahhütleri (commitments), ekiplerin üzerine yüklenen iş yığınları oluşturuyor ve bu yığınlar da sistemdeki iş miktarını (L) artırarak, her bir iş kaleminin tamamlanma süresini uzatıyor. Continuous flow gibi akış odaklı sistemler ise yürütülen iş (WIP) limitleri koyarak L değerini düşük tutar ve böylece teslim süresini (W) minimize eder.[15] Üstelik bu sprint yapısı, mevzu bahis akışı her iki haftada bir sıfırlayarak ve yeni bir yığın işi sisteme sokar böylece sistemde kararsızlık daha da çoğalır neticesinde oluşan darboğazlarla (bottleneck) başa çıkılamaz hale gelir.

Kuyruk Teorisi Parametresi	Sprint Uygulaması	Yaşanan Etki
WIP - Work In Progress (Yürütülen İş)	Sprint Backlog (Yüksek L)	Lead Time'da Artış (Teslimat Süresi Gecikmesi)
Kapasite Kullanımı	%100 Doluluk Hedefi	Bekleme Sürelerinde Eksponansiyel Maksimizasyon
Batch Size (Yığın Boyutu)	Sprint Büyüklüğü	Hata Tespitinde Gecikme ve Yüksek Risk
Değişkenlik (Variability)	Sprint Sonu Teslimat Baskısı	Kalite Kaybı, Teknik Borç Birikimi

Küçük yığınlarla (small batches) çalışmak, geri bildirim döngüsünü hızlandırır ve hataların maliyetini düşürür.[16] Sprint, süresiyle sınırlı olsa da, aslında bu sürenin kendisini bir yığın boyutu olarak kabul eder. Bir hatanın sprintin ilk gününde yapılması durumunda, bu hatanın tespiti çoğu zaman sprint sonundaki test aşamasına kadar sarkar, bu da hatalı bir öğrenme gecikmesi yaratır.

Bilişsel Yük ve Yazılım Mühendisi Psikolojisi: Flow State

Yazılım geliştirme, derin odaklanma ve karmaşık zihinsel modellerin inşasını gerektiren bir disiplindir. Psikolojide akış hali (flow state)[17] olarak bilinen bu yüksek üretkenlik durumu, sürekli kesintilere maruz bırakıldığında, mühendislik performansı dramatik şekilde düşer.[18] Scrum'ın beraberinde getirdiği günlük stand uplar, groomingler, planlamalar, refinementlar, retrospektifler, bu akışı düzenli aralıklarla bölen adeta birer ritüel silsilesi haline gelmiştir. [19]

Gloria Mark ve UC Irvine isimli iki araştırmacı yayınladıkları bir makalede, sürekli bölünen insanların, kaybettikleri zamanı telafi etmek için daha hızlı çalışmaya başladıklarını, ancak bunun bedelinin çok daha yüksek bir stres, hüsran ve zihinsel yorgunluk olduğunu iddia etmektedir.[20] Bunun sonucu, sistem kalitesini gözetmekten ziyade sadece günü kurtarmaya ve önündeki görevi bir an önce tamamlamaya

[14] Geleneksel bakış, kodu canlıya almak (markete gitmek) her zaman zor, pahalı ve riskli bir işti, bugün de öyle, yarın da öyle olacak. Bu bizim değiştiremeyeceğimiz bir doğa kanunudur şeklinde yorumlar. Modern bakış ise markete gitmenin zor olması bir doğa kanunu değildir, eski teknolojilerin sorunudur der. Modern bakışa göre otomasyon kurduğumuzda, o marketi evimizin mutfağına bağlayan bir taşıma bandı inşa etmiş oluruz. Böylece markete gitmenin eforunu (İşlem Maliyetini / Transaction Cost) neredeyse sıfıra indiririz.

[15] Kanban vs. Scrum: 7 Differences You Must Know, Businessmap, https://businessmap.io/kanban-resources/kanban-software/kanban-vs-scrum-software

[16] Impact of Deployment Frequency and Batch Size on Software Quality, Aviator, https://www.aviator.co/blog/impact-of-deployment-frequency-and-batch-size-on-software-quality/

[17] https://www.headspace.com/articles/flow-state

[18] Context Switching Cost for Developers: Research & Data, Super Productivity, https://super-productivity.com/blog/context-switching-costs-for-developers/

[19] Shape Up vs Scrum: How Six-Week Cycles Outperform Sprints for Engineers, Hillia Blog, https://www.hillia.app/blog/shape-up-vs-scrum

[20] https://ics.uci.edu/~gmark/chi08-mark.pdf

odaklanan yazılım ekipleridir. Günde sadece 10 dakika sürmesi öngörülen ama en az yarım saat alan bir daily scrum toplantısı, hazırlık süreci ve sonrasındaki odaklanma kaybıyla birlikte aslında bir geliştiricinin gününden en az 1 saatlik verimli çalışma vaktini çaldığını düşünün. Haftalık planlama, groomingler, vb toplantılar da eklendiğinde, geliştiricilerin vaktinin %15 ile %20'si sadece çalışma hakkında konuşmaya (work about work) harcanıyor denebilir. Bu kayıplardan dolayı estimate ederken hiç bunları dikkate almamış geliştiricinin sprint sonu yaklaştıkça üzerindeki artan teslimat baskısı onun doğruyu yapmak yerine sprinti bitirmek için kestirme yollara sapmasına neden olur. Bu da uzun vadede sistemin bakımını imkansız hale getiren ve business biriminin anlamakta en çok zorlandığı olgu olan teknik borcun esasında birincil kaynağıdır. Hal böyleyken modern teknoloji devleri (Google, Amazon, Netflix gibi), ekiplerine otonomi vererek ve gereksiz ritüelleri ortadan kaldırarak, mühendislerin derin çalışma sürelerini maksimize etmeyi hedeflediklerini iddia etmektedirler. [21]

DORA Metrikleri ve Elite Performans

DevOps Research and Assessment (DORA) grubu, on yılı aşkın süredir yüksek performanslı teknoloji ekiplerini incelemekte ve Elite statüsündeki ekiplerin ortak özelliklerini dört temel metrikle tanımlamaktadır:

1. Yayına Alma Sıklığı (Deployment Frequency)
2. Değişiklik Teslim Süresi (Lead Time for Changes)
3. Değişiklik Başarısızlık Oranı (Change Failure Rate)
4. Hizmet Kurtarma Süresi (Time to Restore Service)

Scrum'ın sprint yapısı, bu metriklerin en önemlileri olan hız göstergeleriyle yapısal bir çatışma halindedir. Elite ekipler günde birden fazla kez yayına alım yaparken, klasik bir Scrum ekibi sprint sonunu beklediği için bu metrikte geride kalır. Aynı şekilde, bir kod değişikliğinin commit edildikten sonra üretime geçmesi (lead time), Scrum'da sprint süresine (genellikle 2 hafta) endekslidir. Bu, modern mühendislik standartlarına göre kabul edilemez derecede yavaş bir hızdır.

Dora Metriği	**Elite Performans**	**Standart Scrum Ekibi**	**Verimlilik Kaybı Nedeni**
Deployment Frequency	Günde 1'den fazla	2 haftada 1 (Sprint sonu)	İterasyon
Lead Time	1 günden az	1 - 4 hafta arası	Batch sizing maliyeti
Change Failure	%0 ile %15 aralığı	Değişken	Sprint sonuna yetiştirme
Time to Restore	1 saatten az	Sonraki Sprint (Genellikle birkaç gün)	Sabit döngü

DORA'nın 2025 raporu, Yapay Zekanın (AI) yazılım geliştirmeyi hızlandırdığını ancak bu hızın sistemdeki darboğazları (bottlenecks) daha belirgin hale getirdiğini vurgulamaktadır.[22] AI ile kod yazım hızı artsa bile, eğer ekip sprinte takılıyorsa, bu hız artışı sadece teknik borcu ve bitmemiş iş (WIP) miktarını artırmaktan başka bir işe yaramaz. Aslında bu düşüncenin altında endüstri ve sistem mühendisliğinin en temel kurallarından biri olan Kısıtlar Teorisi (Theory of Constraints) yatar. Bu teoriye göre, bir sistemdeki en yavaş aşamayı (darboğazı) çözmeden, diğer aşamaları hızlandırmak toplam çıktıyı artırmaz, sadece o darboğazın önünde biriken envanteri (WIP) devasa boyutlara ulaştırır. Yapay zeka sadece kod yazım (development) sürecini hızlandırır. Yani otoyolu 2 şeritten 5 şeride çıkarır. Ancak scrum'ın dayattığı ritüeller yolun sonundaki tek gişelik bir geçiş noktası gibidir. Kod ne kadar hızlı yazılırsa yazılsın, o gişeden geçemedikçe sadece trafik sıkışıklığı (WIP) yaratılmış olacaktır. Ayrıca yapay zeka çok hızlı kod üretse bile bu kodun doğru, güvenli ve mimariye uygun olup olmadığını anlamanın tek yolu onu test etmek ve çalışan yapıya entegre etmektir. Sprint mantığında bu kodlar büyük yığınlar (batches) halinde biriktirildiği için, yapay zekanın yaptığı sistemsel bir hata veya zafiyet anında değil, günler sonra (test aşamasında veya sprint review'da) fark edilir. Geri bildirim geciktikçe, o hatalı kodun üzerine yeni kodlar yazılır ve bu durum katlanarak teknik borca (technical debt) dönüşür. [23]

Planlama Yanılgısı

Yazılım geliştirme, doğası gereği keşfe dayalı, stokastik[24] ve doğrusal olmayan bir süreçtir. Bir problemin çözümü, karşılaşılan bilinmeyenlere (unknown unknowns) göre 3 saat de sürebilir, 3 gün de. Ancak scrum, bu belirsizliği katı zaman aralıklarına (timebox) indirgemeye çalışarak mühendisliğin bu temel gerçeğiyle daha en baştan çelişir.

İnsan beyni, gelecekteki karmaşık görevleri tahmin ederken sistemsel engelleri, bağlam değişimlerini (context switch), teknik borçları göz ardı eder ve işlerin sadece en iyimser senaryoda (happy path) ilerleyeceği varsayımıyla hareket eder. Buna planlama yanılgısı denir.[25] Böyle bir bilişsel önyargının (cognitive bias) var olduğunu bile bile, belirsiz ve keyfice yapılan tahminleri (estimation) kesin birer taahhüt (commitment) gibi kabul etmek başlangıçta belirttiğimiz stokastik süreci deterministik bir zaman dilimine sığdırma çabasından başka bir şey değildir. Bu, continuous delivery'nin savunduğu teslimat bir akıştır ilkesini reddetmekten başka bir şey de değildir.

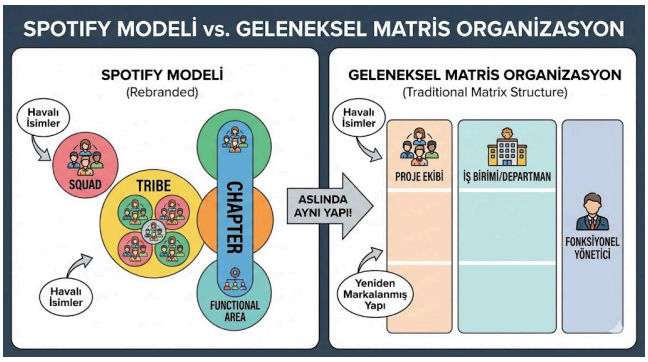
Sonuç olarak sprintler, ekipleri kaçınılmaz bir mini waterfall modeline zorlar. Her iki haftada bir analiz, planlama, kodlama ve test aşamalarından geçilir. Bu döngü, büyük projeleri yönetilebilir parçalara bölüyor gibi görünse de, gerçek dünyadaki iş ihtiyaçları bu yapay takvime uymaz. Eğer bir özelliğin doğal geliştirme süresi 2.5 hafta ise, planlama yanılgısı ile 2 haftaya sığdırılmaya çalışılan bu iş için scrum takımı iki yoldan birini seçmek zorunda kalır. Ya işi anlam bütünlüğünü bozacak şekilde yapay olarak ikiye bölmek ya da kaliteyi düşürüp teknik borç yaratarak o sürenin içine sığdırmak.

Organizasyonel Ölçeklenme ve Spotify Modeli Yanılsaması

Birçok organizasyon, scrum'ın hantallığından kurtulmak için spotify Modeli gibi daha otonom görününen yapılara yönelmiştir. Ancak işin komik tarafı spotify'ın kendisi bile bu modeli tam olarak uygulamadığını ve modelin aslında o dönemdeki spesifik ihtiyaçlara verilen deneysel bir yanıt olduğunu belirtmektedir.[26] Spotify modelindeki squad, tribe, chapter gibi havalı isimler, aslında geleneksel matris organizasyon yapısının yeniden markalanmasıdır. [27] Scrum'ın katı rollerinden (Scrum Master, Product Owner) kaçan ekipler, bu yeni modelde de hesap verebilirlik boşluklarıyla (accountability voids) karşılaşmışlardır.

[21] https://blog.pragmaticengineer.com/project-management-at-big-tech/
https://www.scrums.com/guides/creating-a-high-performance-engineering-culture
[22]https://cloud.google.com/blog/products/ai-machine-learning/announcing-the-2025-dora-report

[23] DORA Report 2025 Summary (State of AI-assisted Software Development), Scrum.org, https://www.scrum.org/resources/blog/dora-report-2025-summary-state-ai-assisted-software-development
[24] belirli bir düzen içerisinde seyretmeyen, belirsizlik unsuru içeren
[25] Daniel Kahneman, planning fallacy, https://en.wikipedia.org/wiki/Planning_fallacy
[26] Spotify Doesn't Use the Spotify Model, Agility 11, https://www.agility11.com/blog/2020/6/22/spotify-doesnt-use-the-spotify-model
[27] The Spotify Model of Scaling: Spotify Doesn't Use It, Neither Should You, Agile Pain Relief, https://agilepainrelief.com/blog/the-spotify-model-of-scaling-spotify-doesnt-use-it-neither-should-you/



Yani aslında teknik mükemmeliyeti (Engineering Excellence) ve sürekli akışı destekleyen bir sistem tasarımı söz konusu değildir.

Modern DevOps dünyasında, ekiplerin otonom olması yetmez, aynı zamanda gevşek bağlı ama sıkı hizalanmış (loosely coupled but highly aligned) olmaları gerekir. Çünkü gevşek bağlılık ekiplerin bürokrasiye ve birbirine takılmadan maksimum hızda üretmesini sağlarken, sıkı hizalanma bu otonom hızın bir kaosa dönüşmeyip şirketin ortak stratejik hedefine doğru ilerlemesini garanti altına alır. Çözüm, otonominin sprintten ziyade, otomatize edilmiş quality gatelere ve TBD gibi teknik standartlara dayandırmaktır.

Yanlışlama (Falsification): Sprint'in Hala Savunulabilir Olduğu Alanlar

Karl Popper'ın yanlışlama ilkesi gereği, Sprint her zaman kötüdür gibi bir iddia bilimsel olarak zayıftır. Scrum'ın yetersizliklerine dair iddiamızı savunurken, bu metodolojinin hangi koşullarda hala en iyi seçenek olabileceğini (boundary conditions) belirlemek, argümanımızı güçlendirecektir.

1. Düşük Olgunluktaki Ekipler (Junior Teams)
TBD ve CD, yüksek disiplin ve üst düzey teknik yetkinlik gerektirir. Otomatize testlerin olmadığı, sürekli entegrasyonun kurulmadığı bir ekipte sprint içerisindeki katı rehberlik ve iki haftalık kontrol noktaları, tam bir kaostan daha iyi olacaktır.

2. Yüksek Dış Bağımlılıklı Projeler
Yazılımın bir donanım veya karmaşık bir tedarik zinciriyle entegre olması gerektiği durumlarda, tüm tarafların aynı ritimle (sprint) hareket etmesi koordinasyonu kolaylaştırabilir. Feature flag kullanmadan continuous flow koşmaya çalışılan ortamlarda (ki bu tavsiye edilmez) bu tür sert bağımlılıklar senkronizasyon sorunları yaşatabilir.

3. Ürün Tanımının Hiç Olmadığı Durumlar
Pazarın ve ihtiyacın tamamen belirsiz olduğu bir ortamda, en azından her iki haftada bir yanlış yolda olduğumuzu görelim yaklaşımı, bir stratejik savunma içgüdüsü olabilir.

4. Büyük Ölçekli Kurumsal Geçişler
Waterfall'dan yeni çıkan bir organizasyon için continuous flow korkutucu olabilir. Scrum, bu geçiş aşamasında bir ara durak veya köprü görevi görerek organizasyona çeviklik disiplinini aşılayabilir.

Bu durumlar bile idealden ziyade, geçici bir çözüm veya zorunlu bir kısıt olduğunu gösterir. Teknik mükemmeliyete ulaşmış, yüksek güvenli (high-trust) ve otonom mühendislik kültürlerinde scrum, gelişim hızını frenleyen bir mekanizmaya dönüşür.

Sonuç: İterasyondan Sürekli Akışa Geçiş

Scrum, bizleri waterfall'un hantallığından kurtararak devrimsel bir rol oynamıştır. Ancak günümüzün DevOps, TBD ve diğer standartları altında, scrum'ın sprint temelli zaman kesitli çalışma mantığı, bir çözümden ziyade bir engele dönüşmüştür. Modern mühendislik ekiplerinin işi yönetmek için esas ihtiyacı, iş akışını optimize edecek teknik altyapıları oluşturmaktır, zaman kesitlerinde işi bölmek değil.

Kuyruk teorisi (Little Yasası), sprint yığınlarının teslimat sürelerini uzattığını kanıtlamaktadır. Bilişsel psikoloji araştırmaları, Scrum ritüellerinin mühendislik akışını sabote ettiğini doğrulamaktadır. DORA verileri, Elite ekiplerin neredeyse sadece sürekli akış ile var olabileceklerini göstermektedir. Bu analitik tablonun karşısında organizasyonların yüzleşmesi gereken temel soru şudur: Çeviklik (Agility) hedefiyle yola çıkıp statik ritüellere körü körüne itaat eden bir yapıya mı dönüştük, yoksa değişime anında tepki verebilen gerçek elit bir takım mı yarattık?

Yazılım yönetimi, insanları yönetmeye odaklanmak yerine sistemi ve akışı tasarlamaya yönelmedikçe, kutsallaştırılmış ezberlerin ve amaçların önüne geçen yöntemlerin içinde savrulmaya mahkumdur.